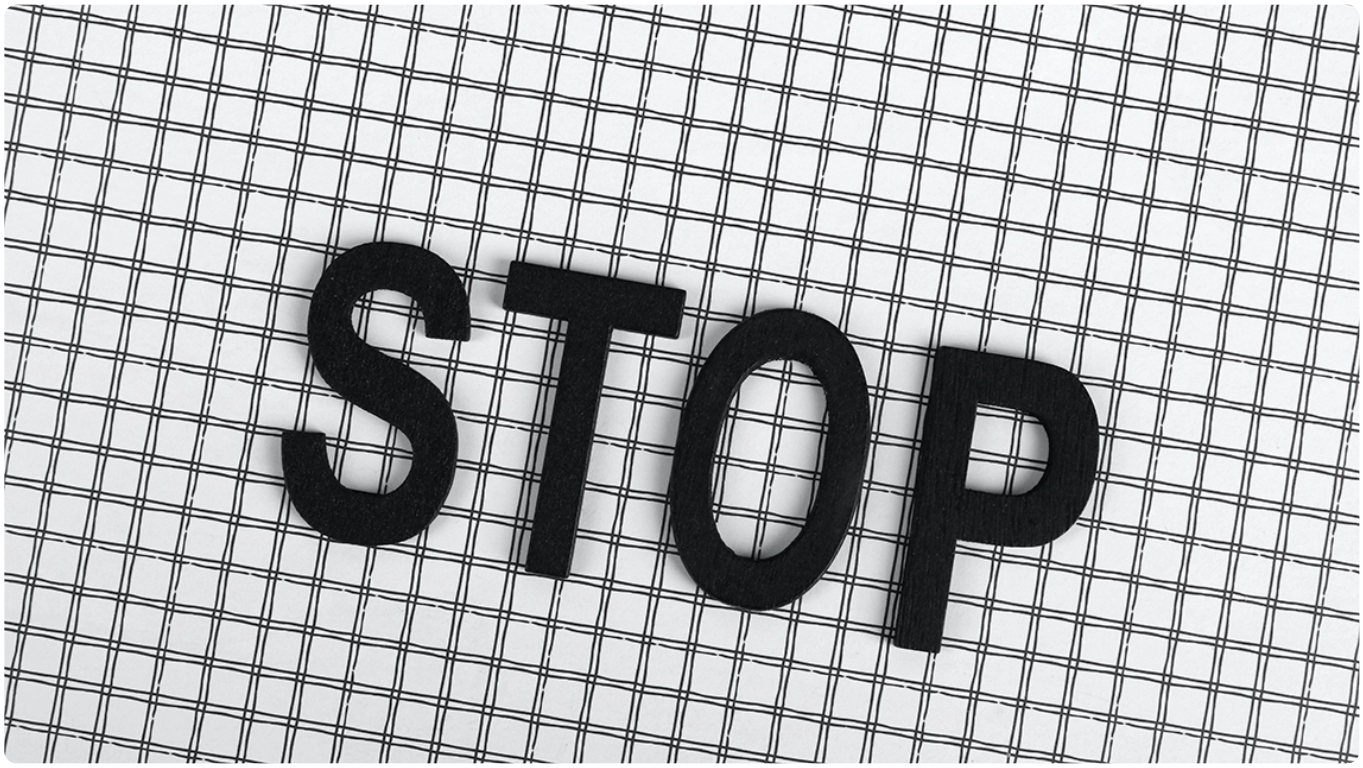
09｜御敌国门外：黑产对抗——防刷和风控

2021-10-15 佘志东

《手把手带你搭建秒杀系统》 课程介绍 >



讲述：佘志东
时长 15:54　大小 14.57M

你好，我是志东，欢迎和我一起从零打造秒杀系统。

经过前面对秒杀业务的介绍，你现在应该清楚，秒杀系统之所以流量高，主要是因为一般使用秒杀系统做活动的商品，基本都是稀缺商品。稀缺商品意味着在市场上具有较高的流通价值，那么它的这一特点，必定会引来一群“聪明”的用户，为了利益最大化，通过非正常手段来抢购商品，这种行为群体我们称之为黑产用户。

他们确实是聪明的，因为他们总能想出五花八门的抢购方式，有借助物理工具，像“金手指”这种帮忙点击手机抢购按钮的；有通过第三方软件，按时准点帮忙触发 App 内的抢购按钮的；还有的是通过抓取并分析抢购的相关接口，然后自己通过程序来模拟抢购过程的。

可不管是哪种方式，其实都在做一件事，那就是**先你一步**。因为秒杀的抢购原则无外乎两种，要么是绝对公平的，即先到的请求先处理，暂时处理不了的，会把你放入到一个等待队列，然后慢慢处理。要么是非公平的，暂时处理不完的请求会立即拒绝，让你回到开始的地方，和大家一起再比谁先到，如此往复，直至商品售完。

因此黑产的方法也很简单，就是想法设法比别人快，发出的请求比别人多，就像在一个赛道上，给自己制造很多的分身，不仅保证自己比别人快，同时还要把别人挤出赛道，确保自己能够到达终点。

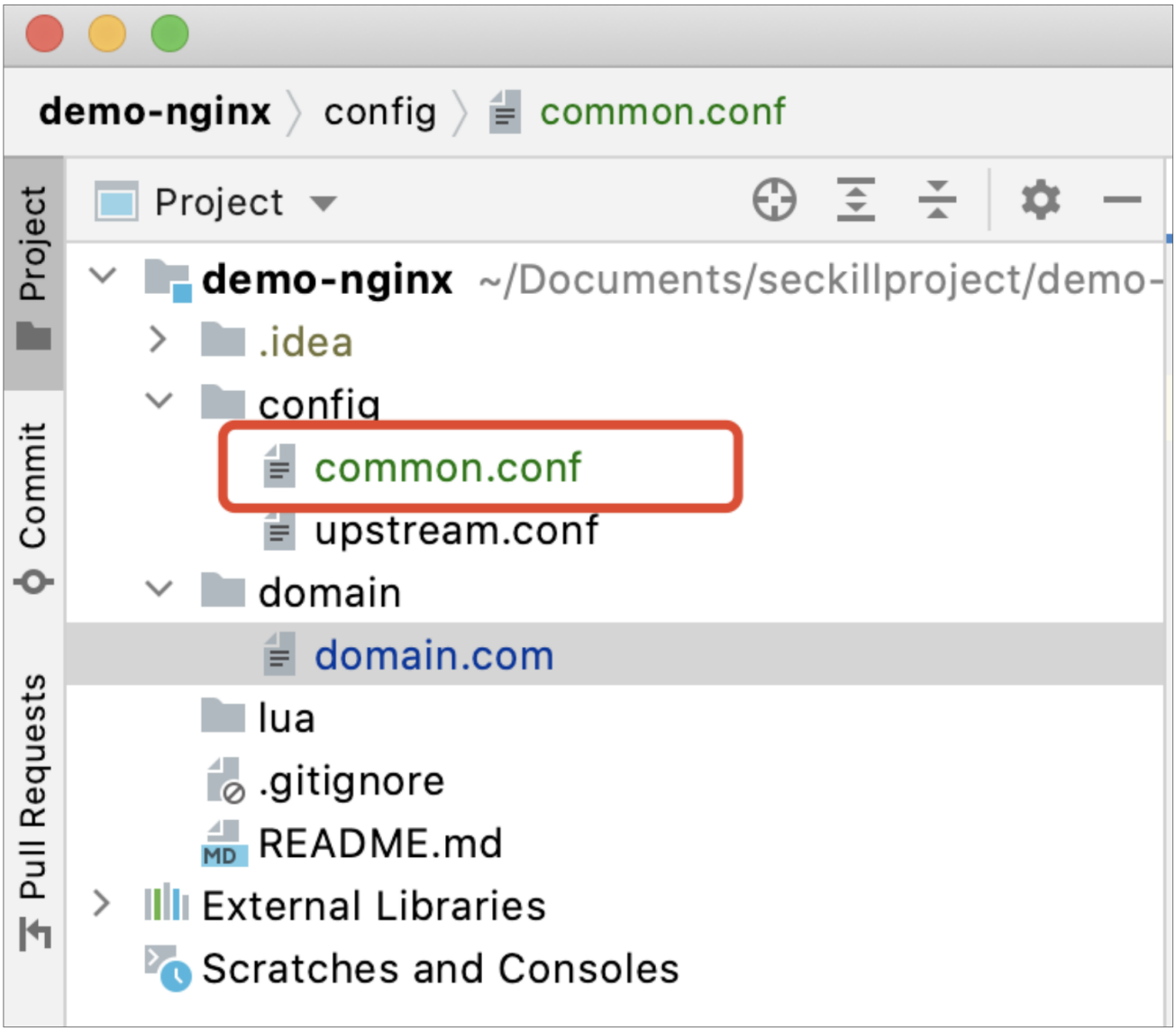
所以黑产对秒杀业务的威胁是巨大的，它不仅破坏了公平的抢购环境，而且给秒杀系统带来了庞大的性能开销，所以我们不能放任黑产流量对系统的肆意冲击，我们必须对抗它。既然黑产流量的特点是比正常流量快且频率高，那么我们也就可以从这两个方面来着手思考对策。

只针对第一个快的特点，其实在活动开始后，进来的流量我们都无法将其定义为非法流量，这个只能借助像风控这种多维度校验，才能将其识别出来，除非它跳步骤。而第二个高频率的特点，同时也是对秒杀系统造成危害最大的一种，我们还是有很多种手段来应对的。所以这节课我就给你介绍几种比较有效且经过实践的防刷方案，它们**专门针对高频率以及跳步奏的非法手段**。

防刷：Nginx 有条件限流

在上节课中我已经介绍过 Nginx 限流的语法了，现在咱们就直接来实践。这里呢，我们是根据用户 ID 来做限流防刷的。

首先我们新建一个通用配置文件 common.conf，用来定义限流规则以及后续一些其他的通用配置，所在位置如下：



同时，不要忘记在 nginx.conf 文件中将该配置文件引入进去，和引入 upstream.conf 一样，如下图所示：

```
worker_processes 1;

error_log logs/error.log error;

events {
    worker_connections 256;
}

http {


        log_format  access  '$remote_addr - $remote_user [$time_local] "$request" $status '
            '"$upstream_addr" "$upstream_status" "$upstream_response_time" userId:"$user_id"';

        lua_package_path "/Users/wangzhangfei5/Documents/seckillproject/demo-nginx/lua/?.lua;;";

        include /Users/wangzhangfei5/Documents/seckillproject/demo-nginx/domain/domain.com;
        include /Users/wangzhangfei5/Documents/seckillproject/demo-nginx/config/upstream.conf;
        include /Users/wangzhangfei5/Documents/seckillproject/demo-nginx/config/common.conf;
}
```

然后，我们在 common.conf 中定义限流规则如下：

common.conf

```
#limit by user
limit_req_zone $user_id zone=limit_by_user:10m rate=1r/s;
```

意为定义了一个名为 limit_by_user 的限流规则，根据用户 ID 来做限流，限流的速率为同一个用户 1 秒内只允许 1 个请求通过，且为该规则申请的内存大小为 10M。

这里的 10M 大概是什么概念呢？可以简单粗略地算下，假如一个 user_id 占用的内存大小为 16 字节，那么 10M 的内存大概可以处理单机 10*1024*1024/16=655360 个请求。

规则配置完毕后，接下来就在我们需要限流的接口引用该规则，这里依然以活动查询接口为例，配置如下：

domain.com

```
server {
        listen 7081;

        error_log logs/domain-error.log error;
        access_log logs/domain-access.log access;
        default_type text/plain;

        #用户ID
        set_by_lua_block $user_id {
           return "zhangsan"
        }

        #活动数据查询
        location /activity/query{
           limit_req zone=limit_by_user nodelay;
            proxy_pass http://backend;
        }

```

其中 nodelay 是被限流后的策略，意为不等待，直接返回。

配置好之后，我们启动 Nginx，通过 URL 进入到商详页（要在活动进行中时）。

荣耀手机特价998，性价比高，最优的选择，不再犹豫，买到即赚到

商品信息

价格：998.00

立即抢购

这时，我们通过鼠标快速地刷新两次页面（点击浏览器中的刷新图标，在 1 秒内完成），来模拟外部的请求，然后看下对应的 access 和 error 日志，结果如下：

通过 domain-access.log 日志可以看到，两次请求，第一次正常返回，第二次返回给客户端 503 的状态码，原因通过 domain-error.log 可以看到，是触发了根据用户 ID 的限流规

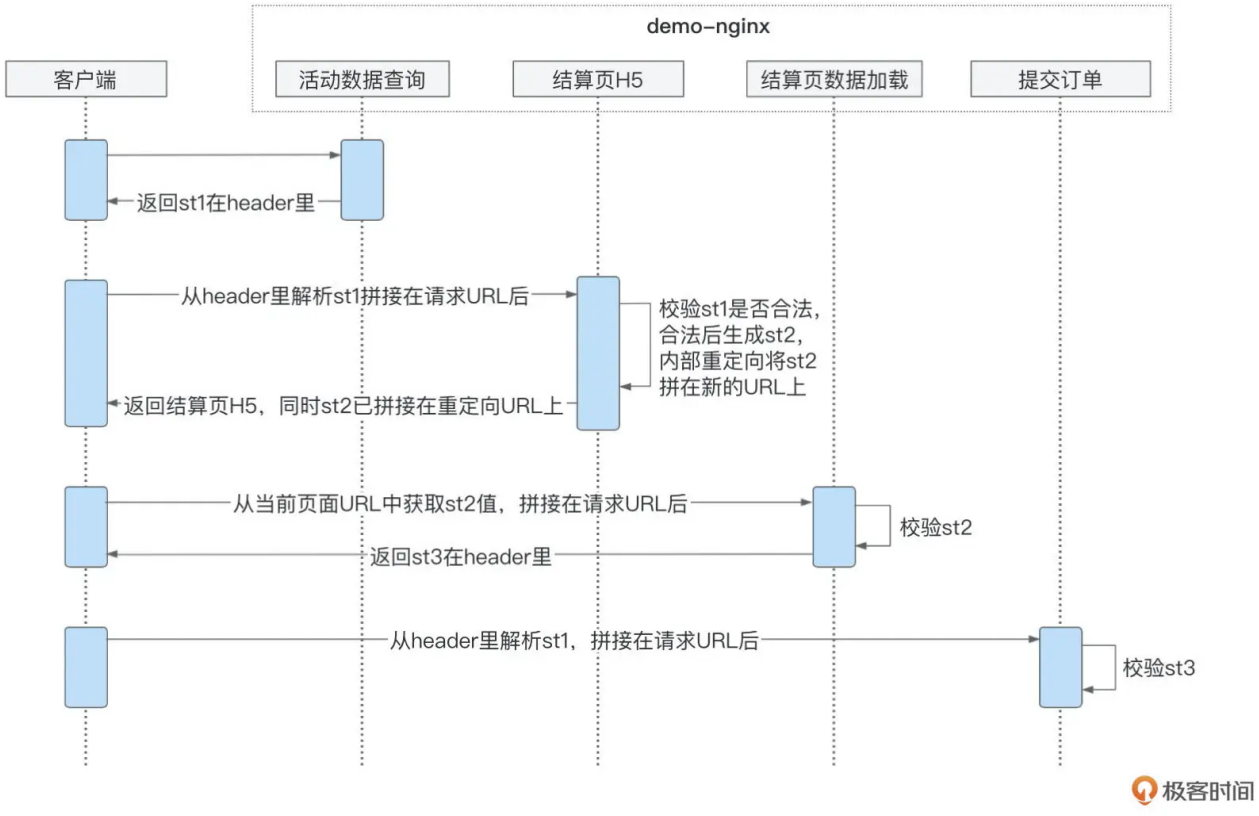
则，这样我们的限流防刷功能就实现了。

以上通过限流的方式来防刷，是非常简单且直接的一种方式，**这种方式可以有效解决黑产流量对单个接口的高频请求**，但要想防止刷子不经过前置流程直接提单，还需要引入一个流程编排的 Token 机制。

防刷：Token 机制

Token 我想你是知道的，一般都是用来做鉴权的。放到秒杀的业务场景就是，对于有先后顺序的接口调用，我们要求进入下个接口之前，要在上个接口获得令牌，不然就认定为非法请求。同时这种方式也可以防止多端操作对数据的篡改，如果我们在 Nginx 层做 Token 的生成与校验，可以做到对业务流程主数据的无侵入。

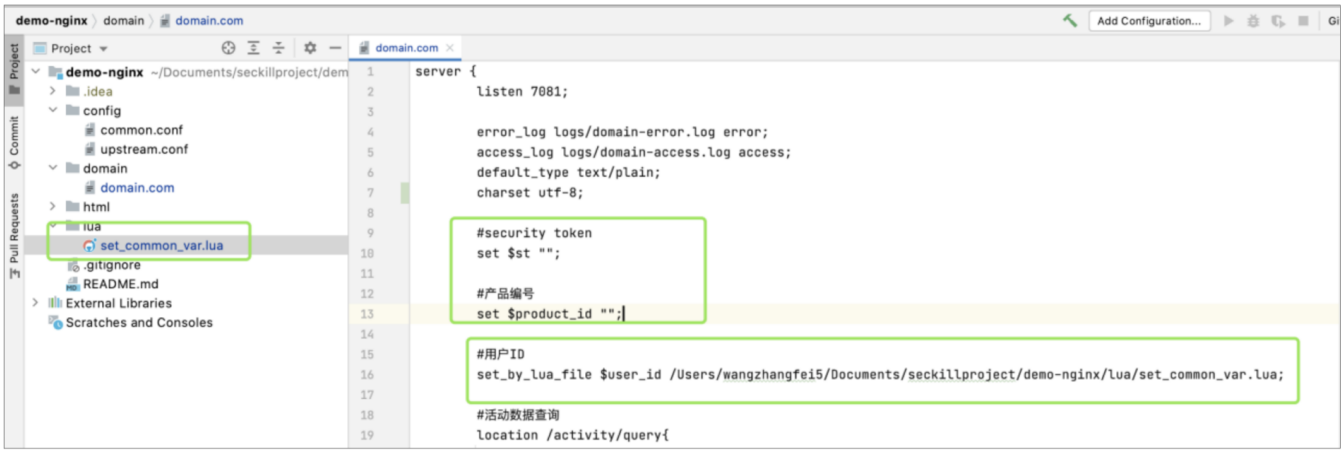
在 Token 机制下，前端与 demo-nginx 中的接口交互时序图如下所示：



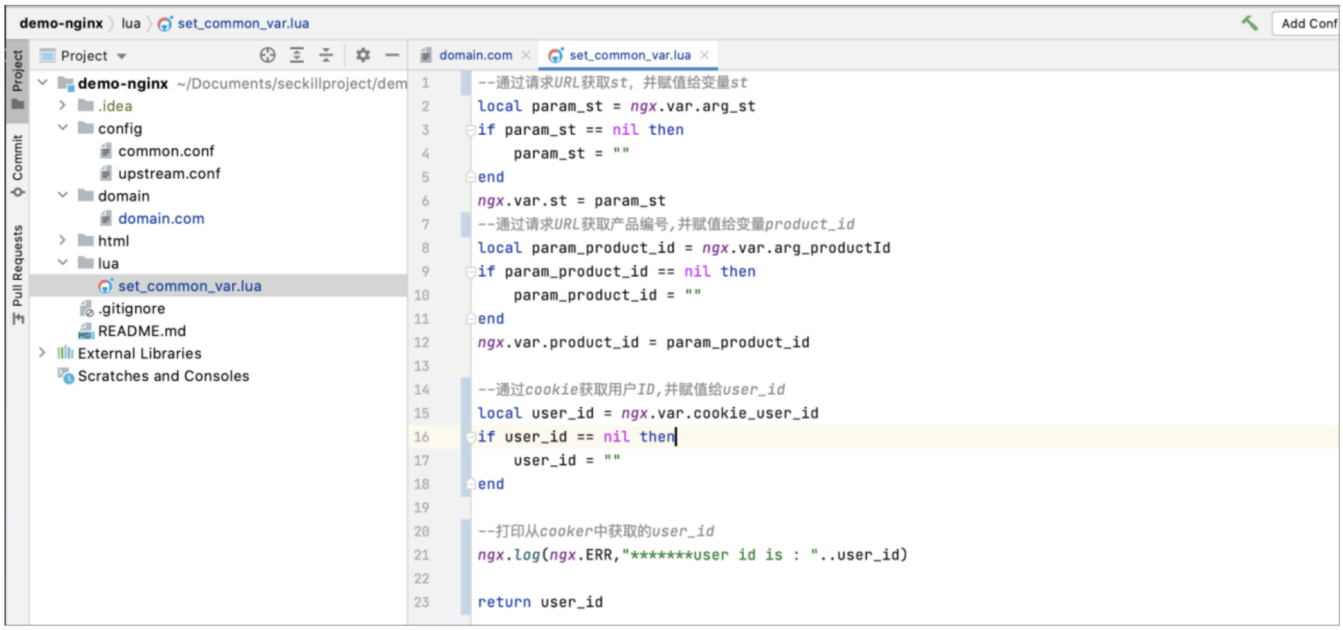
现在我们就按照对应的时序，依次给 4 个接口增加上 Token 相关的生成与校验功能。

在这之前，我们为了更真实地获取用户 ID，需要在 demo-web 中新增加个登录功能，模拟将 user_id 放到 cookie，这样之后的每次请求，我们就直接从 cookie 中获取 user_id 即可。

同时根据我们的设计，有几个主要参数是每次接口请求都必须要校验的，那就是用户 ID、产品编号，还有新的 Token，所以我们就定义了一个统一的解析方法，其中用户 ID 从 cookie 中解析，产品编号和 st 从请求的 URL 中解析，如下图所示：



set_common_var.lua 主要就是负责参数的解析，并给对应的变量做赋值，如下图所示：



做完之后，我们开始改造活动数据查询接口，改造后如下：

domain.com

```
server {
        listen 7081;

        error_log logs/domain-error.log error;
        access_log logs/domain-access.log access;
        default_type text/plain;
        charset utf-8;

        #security token
        set $st "";

        #产品编号
        set $product_id "";

        #用户ID
        set_by_lua_file $user_id /Users/wangzhangfei5/Documents/seckillproject/demo-nginx/lua/set_common_var.lua;

        #活动数据查询
        location /activity/query{
            limit_req zone=limit_by_user nodelay;
            proxy_pass http://backend;
            #设置返回的header，并将security token放在header中
            header_filter_by_lua_block{
               ngx.header["st"] = ngx.md5(ngx.var.user_id.."1")
               ngx.header["Access-Control-Expose-Headers"] = "st"
            }
        }

```

这里通过 header_filter_by_lua_block 指令，在返回的 header 里增加流程 Token。这里 st 的生成只是简单地将用户 ID+ 步骤编号做了 MD5，生产上需要更严格一些，需要加入商品编号、活动开始时间、自定义加密 key 等，这样前端通过解析请求响应 header，就可以拿到 st 了，然后将其拼在请求结算页 H5 的 URL 后即可。

然后再对结算页 H5 的 location 做改造，改造后如下图所示（当然这里可以将 rewrite_by_lua 的内容放到 file，看起来会更整洁一些）：

domain.com

```
        #进结算页页面（H5）
        location /settlement/prePage{
            default_type text/html;
            rewrite_by_lua_block{
                --校验活动查询的st
                local _st = ngx.md5(ngx.var.user_id.."1")
                --校验不通过时，以500状态码，返回对应错误页
                if _st ~= ngx.var.st then
                  ngx.log(ngx.ERR,"st is not valid!!")
                  return ngx.exit(500)
                end
                --校验通过时，再生成个新的st，用于下个接口校验
                local new_st = ngx.md5(ngx.var.user_id.."2")
                --ngx.exec执行内部跳转,浏览器URL不会发生变化
                --ngx.redirect(url,status) 其中status为301或302
                local redirect_url = "/settlement/page".."?productId="..ngx.var.product_id.."&st="..new_st
                return ngx.redirect(redirect_url,302)
            }
            error_page 500 502 503 504 /html_fail.html;
        }

        #进结算页页面（H5）
        location /settlement/page{
            default_type text/html;
            proxy_pass http://backend;
            error_page 500 502 503 504 /html_fail.html;
        }
```

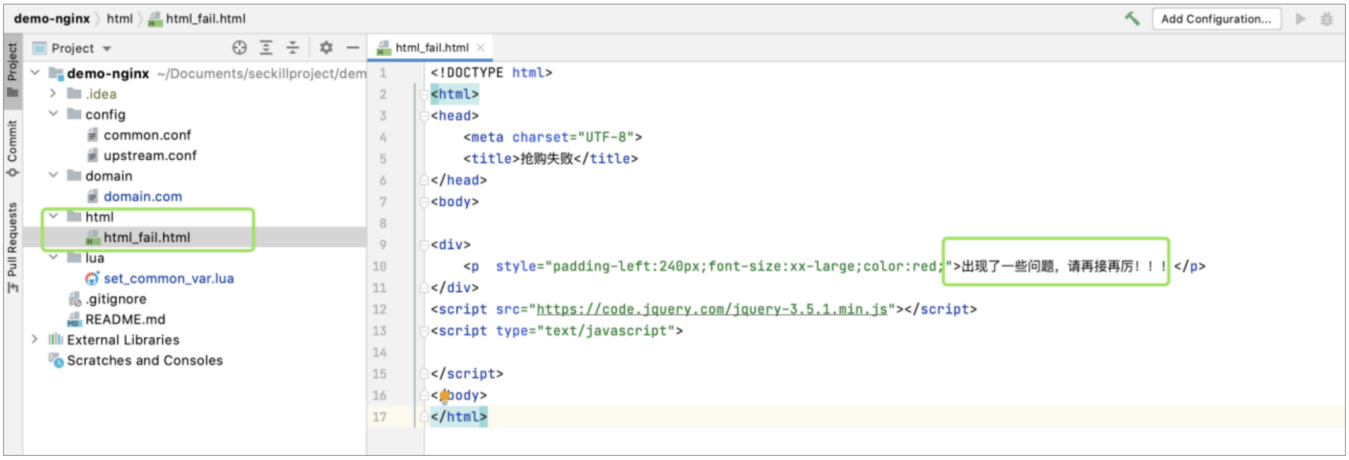
结算页 H5 的改造点，就是将之前的一个 location 拆成了 2 个了，增加了 /settlement/prePage，这样点击立即抢购时就可以直接调用这个接口。

这里你可能会有疑问，**这里为什么没有选择将 st 放在 header 中返回呢？**因为和活动数据查询接口不同的是，这个接口返回的是 HTML，上个接口返回的是 JSON，所以选择了重定向的方式，这样浏览器就可以获得到新的 st 了，具体代码如上，在 rewrite_by_lua_block 配置中使用 ngx.redirect() 方法来实现重定向。

同时这里再给你介绍下，如果业务校验不通过，想终止整个请求流程，可以通过 ngx.exit（状态码）来实现，这样就可以将对应状态码返回给前端了。但如果想要做得更友好些，当系统内部异常或者是后端服务器异常时，我们可以指定返回的内容。这就需要通过 error_page 指令来实现，意为出现不同的错误码，我们会转到不同的 location 去做处理。如果是 H5 请求，那就返回对应的错误提示页，如果是 JSON 请求，也可以返回自定义的 JSON 数据，如下图所示：

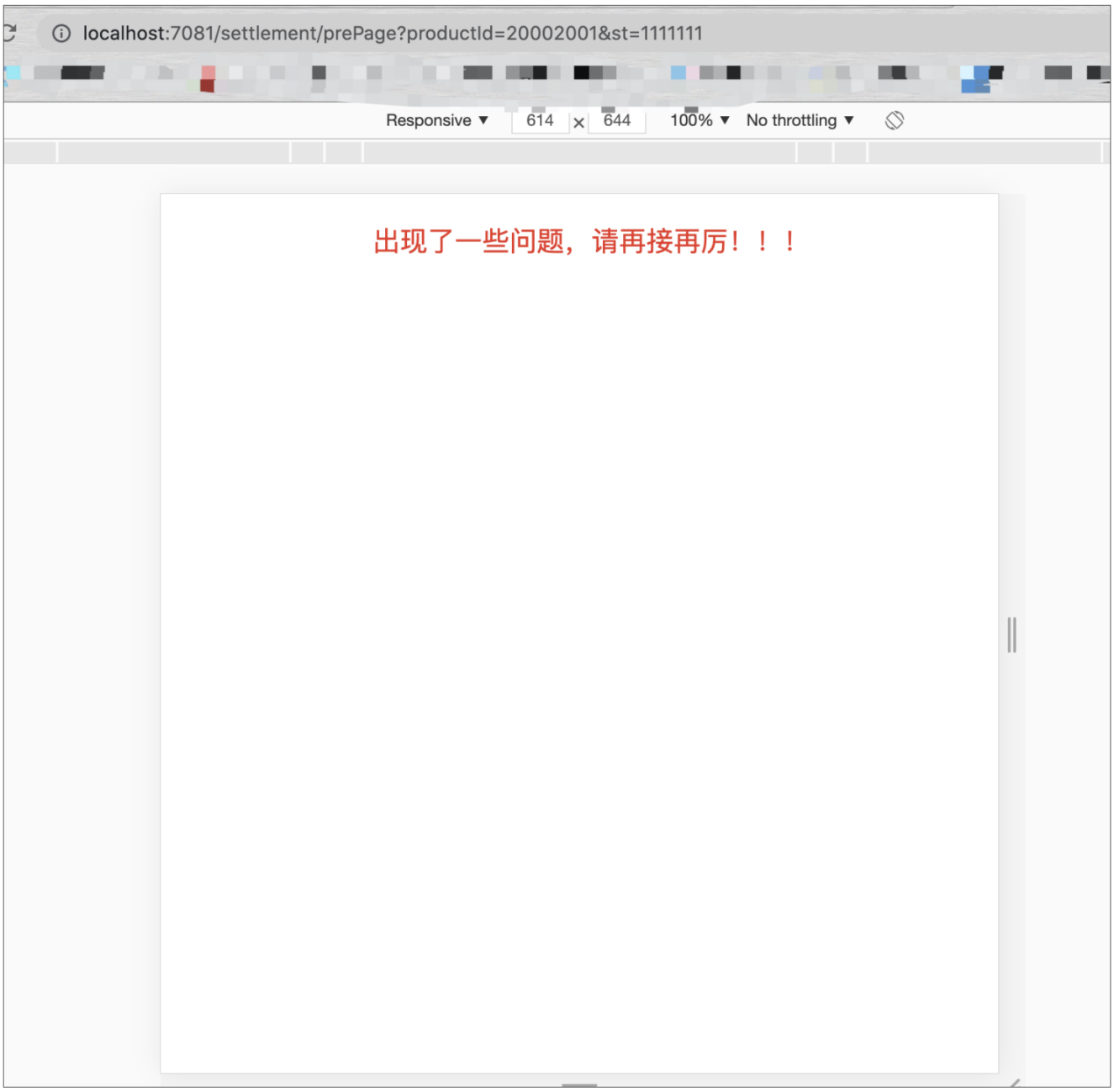
```
        #错误页
        location = /html_fail.html {
            default_type text/html;
            root /Users/wangzhangfei5/Documents/seckillproject/demo-nginx/html;
        }

        #以"@"开头定义的location，是内部接口，外部无法访问
        location @json_fail {
            default_type application/json;
            return 200 '{"code":"200001","message":"nginx intercept!!!"}';
        }
```

上图就是配置了两个异常处理 location，其中 html_fail.html 的位置和内容如下：



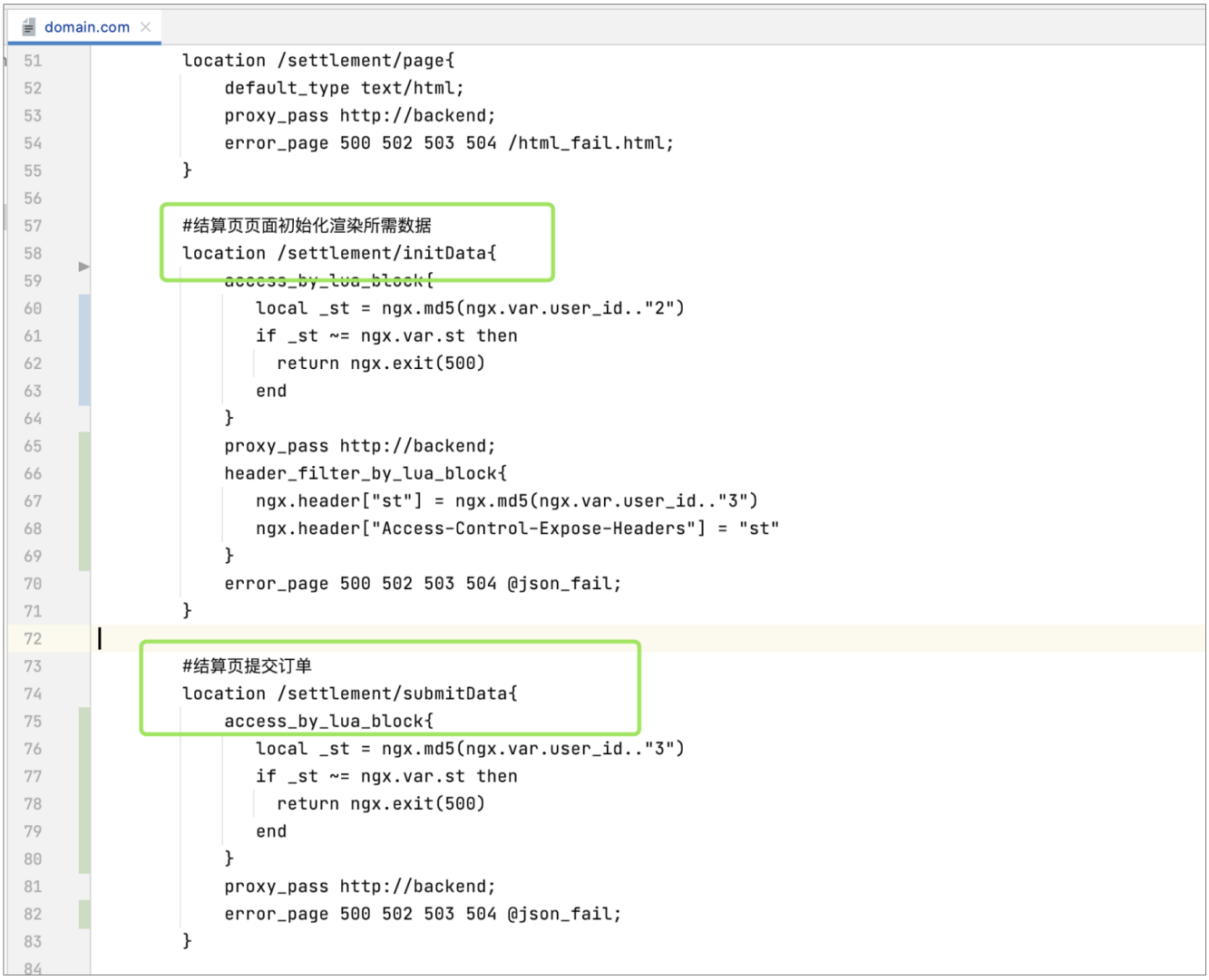
那么做完了上面那么多工作，现在我们就来验证下针对活动数据查询接口和结算页 H5 接口的顺序编排是否生效。

我们启动 Nginx，并且进入到商详页，点击立即抢购，正常流程是可以进入到结算页的，这里就不展示了，那如果我们不经过商详页，直接通过 URL 访问结算页 H5 接口（不带 st 或者带错误的 st），效果会如何呢？看下图：



出现了我们配置的错误提示页，同时日志中出现了 st 校验不通过的提示：

那么使用同样的机制，我们可以把剩下的接口功能给补上，因为做法一样，这里就不多说了，直接上最终改造后的代码，如下：



```
domain.com
51        location /settlement/page{
52            default_type text/html;
53            proxy_pass http://backend;
54            error_page 500 502 503 504 /html_fail.html;
55        }
56
57        #结算页页面初始化渲染所需数据
58        location /settlement/initData{
59            access_by_lua_block{
60               local _st = ngx.md5(ngx.var.user_id.."2")
61               if _st ~= ngx.var.st then
62                 return ngx.exit(500)
63               end
64            }
65            proxy_pass http://backend;
66            header_filter_by_lua_block{
67               ngx.header["st"] = ngx.md5(ngx.var.user_id.."3")
68               ngx.header["Access-Control-Expose-Headers"] = "st"
69            }
70            error_page 500 502 503 504 @json_fail;
71        }
72
73        #结算页提交订单
74        location /settlement/submitData{
75            access_by_lua_block{
76               local _st = ngx.md5(ngx.var.user_id.."3")
77               if _st ~= ngx.var.st then
78                 return ngx.exit(500)
79               end
80            }
81            proxy_pass http://backend;
82            error_page 500 502 503 504 @json_fail;
83        }
84
```

到这，防刷的 Token 机制就介绍完了，**这种机制可以有效防止黑产流量跳过中间接口，直接调用下单接口**。通过该机制 +Nginx 有条件限流机制，可以有效拦截大部分场景下的刷子流量。但如果我们还想再严格一些，对于黑产不仅仅是想拦截过多的非法请求，而是想全部拦截，那有没有什么办法呢？有，咱们继续学习下黑名单机制。

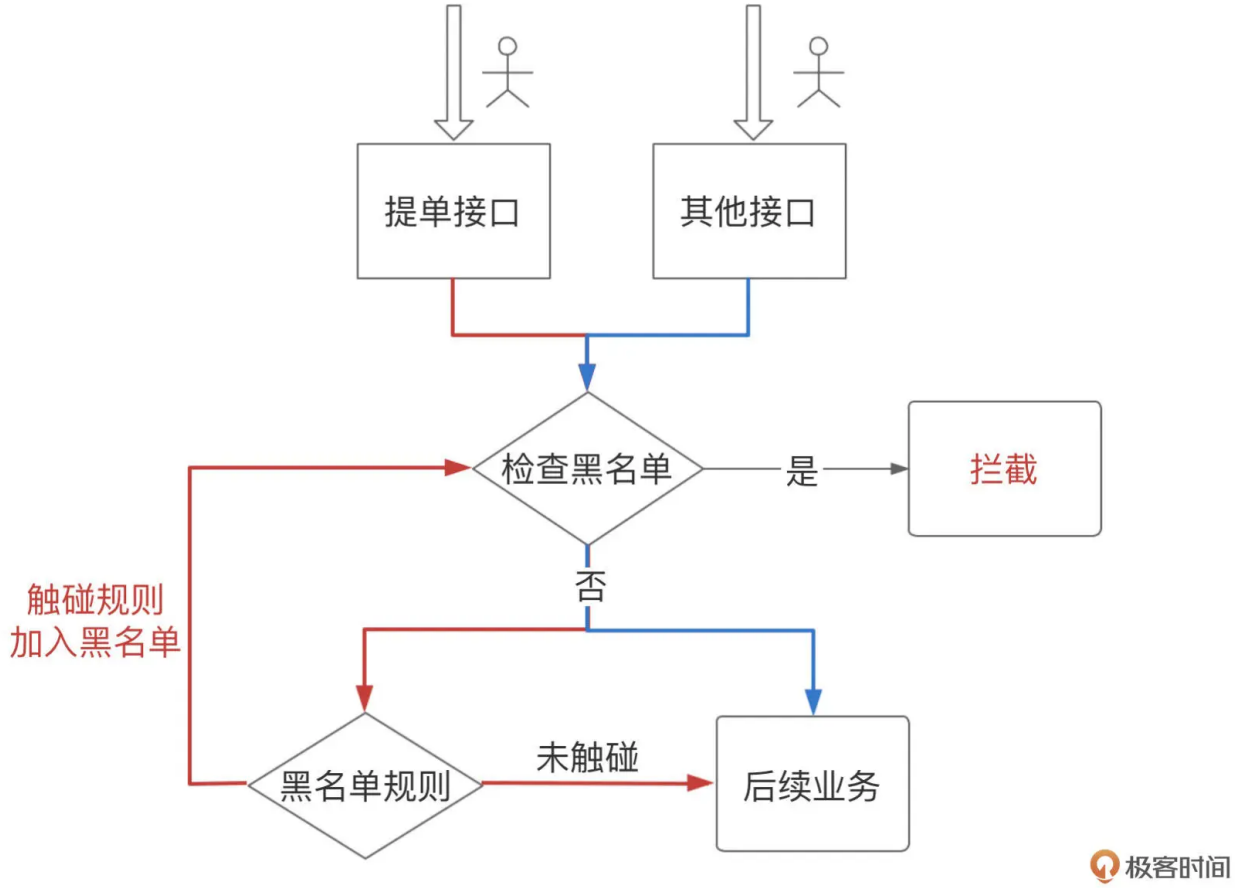
防刷：黑名单机制

黑名单机制分为本地黑名单和集群黑名单两种，接下来我们会重点介绍本地黑名单。该机制顾名思义，就是通过黑名单的方式来拦截非法请求的，但我们的核心问题是黑名单从哪里来呢？

总体来说，有两个来源：一个是从外部导入，可以是风控，也可以是别的渠道；而另一个就是自力更生，自己生成自己用。

前面介绍了 Nginx 有条件限流会过滤掉超过阈值的流量，但不能完全拦截，所以索性就不限流，直接全部放进来。然后我们自己实现一套“逮捕机制”，即利用 Lua 的共享缓存功能，去统计 1 秒内这个用户或者 IP 的请求频率，如果达到了我们设定的阈值，我们就认定其为黑产，然后将其放入到本地缓存黑名单。黑名单可以被所有接口共享，这样用户一旦被认定为黑产，其针对所有接口的请求，都将直接被全部拦截，实现刷子流量的 0 通过。

这里以提单做逮捕入口示例，大致流程如下：



其中红线表示黑名单的校验与生成路径，蓝线表示普通接口的黑名单校验路径。那么下面我们就根据这个设计思路，去具体实现它。

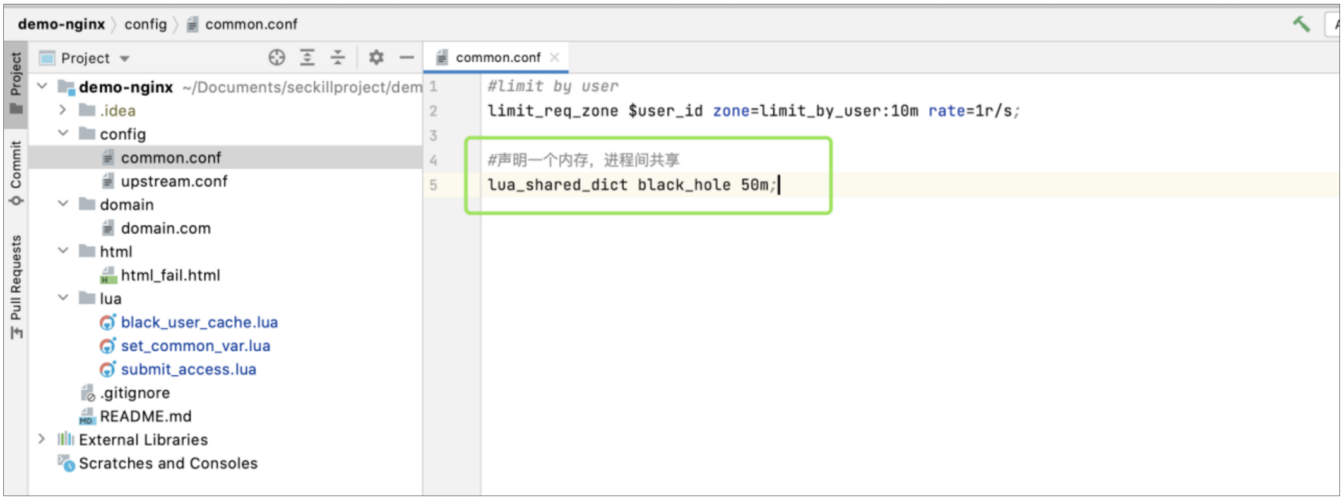
首先新建两个 Lua 文件，一个叫 submit_access.lua，另一个叫 black_user_cache.lua。然后修改提单接口的 access_by_lua_* 指令，将其替换成 access_by_lua_file，并引用 submit_access.lua，如下图所示：

其中 black_user_cache.lua 主要用来写和黑名单相关的本地缓存逻辑，包括黑名单的检查以及黑名单的过滤生成。其中黑名单的过滤方法是核心逻辑，这里主要是根据传入的 key，利用 Lua 缓存的自动失效机制，来统计一段时间内的请求数，在达到了我们设定的阈值后，便将其放到本地缓存中，并提供针对 key 的查询功能，如下图所示：

black_user_cache.lua

```lua
local _CACHE = {}
-- 共享内存区域，用于统计黑名单以及保存黑名单
local hole = ngx.shared.black_hole
-- 请求计数key的前缀
local count_prefix = "co_"
-- 黑名单key的前缀
local black_prefix = "bl_"
-- 过滤请求，如果没有触碰黑名单规则，则返回true,反之则返回false
function _CACHE.filter(key)
    --  参数1 key, 参数2 步长，参数3 如果key不存在时的初始化值 ,参数4 初始化值的失效时间
    local after_count = hole:incr(count_prefix..key, 1, 0, 1)
    -- 如果为空，则是异常了，这里不做拦截，防止误杀
    if not after_count then
        return true
    end
    -- 判断1秒内的请求频率，如果大于设定阈值，则加入黑名单
    if after_count > 1 then
        ngx.log(ngx.ERR,key.." was caught ! ! ! ")
        -- 存入本地cache,有效期15秒
        local suc, err = hole:set(black_prefix..key,1,15)
        if not suc
        then
            ngx.log(ngx.ERR,key.." set to cache fail : "..err)
        end
        return false
    end
    return true
end
-- 校验是否合法，如果在黑名单，则返回false，如果不在，则返回true
function _CACHE.check(key)
    local value = hole:get(black_prefix..key)
    if not value then
        return true
    end
    return false
end
return _CACHE
```

这里引入了 Lua 本地缓存的概念，所以在使用前我们需要先申请一个内存块。我们将申请的地方放在了 common.conf 中，其中 black_hole 就是我们申请的内存名称，大小为 50M，如下图所示：



底层方法写好后，我们在 submit_access.lua 中引入 black_user_cache.lua，实现黑名单的判断、过滤以及前面的 st 校验功能：

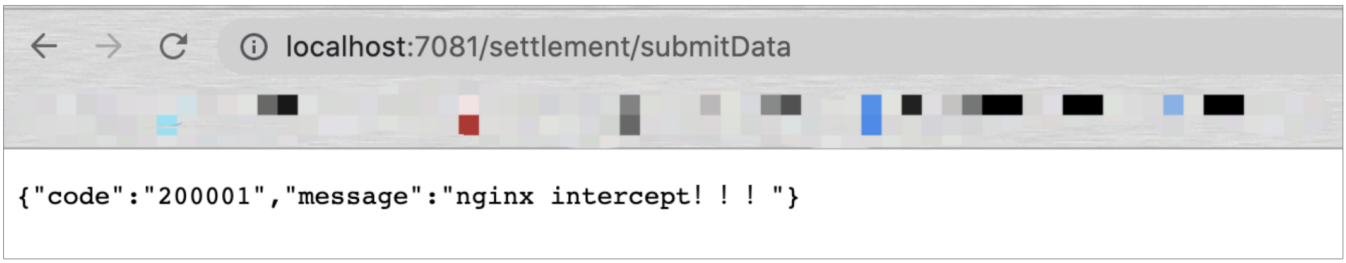
submit_access.lua

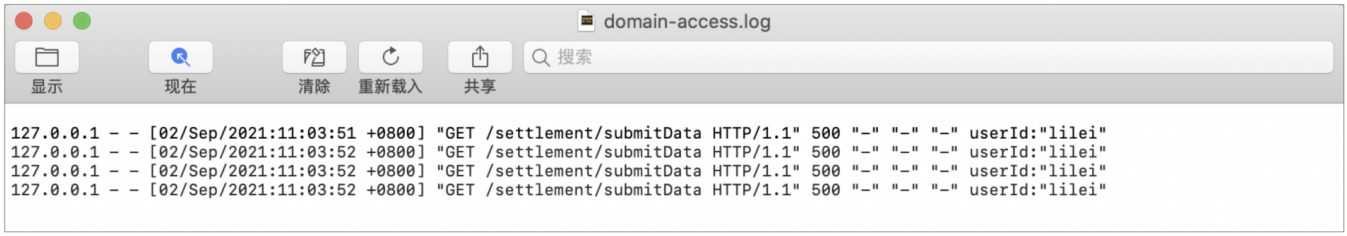
```lua
-- 引入黑名单相关的缓存
local black_cache = require "black_user_cache"

local user_id = ngx.var.user_id

-- 统计key，就用user_id
local key = user_id

-- 校验是否在黑名单
local flag = black_cache.check(key)
if not flag then
    ngx.log(ngx.ERR,key.." is in black list!")
    return ngx.exit(500)
end

-- 校验是否触碰黑名单生成规则
local ff = black_cache.filter(key)
if not ff then
    return ngx.exit(500)
end

-- 校验st是否合法
local _st = ngx.md5(user_id.."3")
if _st ~= ngx.var.st then
    ngx.log(ngx.ERR,user_id." st is invalid!")
    return ngx.exit(500)
end
```

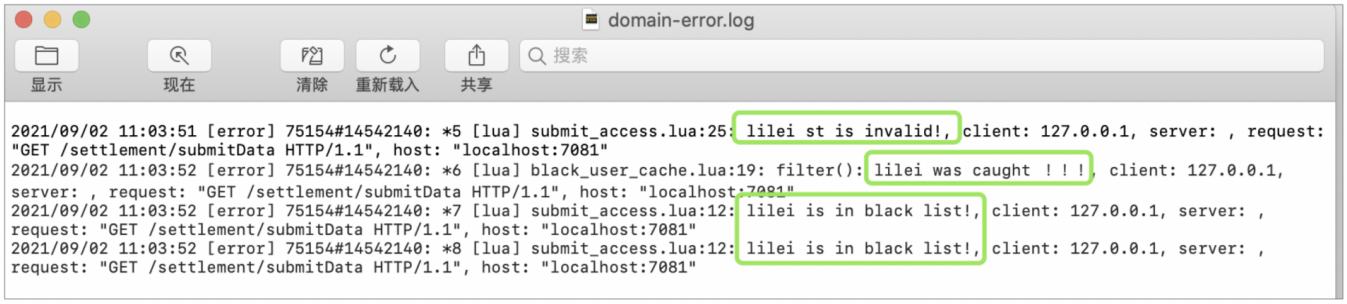
一切完成后，我们来测试一下，看是否生效。我们可以直接在浏览器访问提单功能，然后模拟 1 秒内请求提单接口超过两次的场景，但在看效果前，我们可以根据上面的逻辑实现，先设想下可能会出现的场景。

首先肯定是每次请求都会返回错误提示，第一次错误是因为我们没有 st；第二次错误是因为我们触发了黑名单规则，然后被加进了黑名单；第三次及以后的请求，错误都是因为已经在黑名单，被直接拦截了。

好的，那么下面就让我们看下页面效果以及 access 和 error 日志是否达到了预期：







可以看到，1 秒内有 4 次请求进来，且响应结果都与我们预期的一样。上面的逻辑中，比如统计的频率，统计的 key，黑产被加入黑名单的有效期等，都可以根据实际业务灵活变化。

本地黑名单机制的优点就是简单、高效。但也正因为基于单机，如果黑产将请求频率控制在 1*Nginx 机器数以内，按请求理想散落的情况下，那么就不会被我们抓到，所以真要想通过频率来严格限制刷子请求，是可以借助 Redis 来实现集群黑名单的。

实现思路和单机的基本一致，就是使用的内存由本地变为了 Redis，当然这也必然会影响接口的响应性能，所以这里我只给出了一个进一步收紧校验的思路，就不做具体教学了，感兴趣的话可以自己尝试下。

风控

以上我们介绍了如何通过防刷的手段与黑产用户对抗。而想要更全面地对抗黑产，我们还需要引入另一个重要的机制，那就是风控。

风控在秒杀业务流程中非常重要，但风控的建立却是非常困难的。成熟的风控体系需要建立在大量的数据之上，并且要通过复杂的实际业务场景考验，不断地做智能修正，才能逐步提高风险识别的准确率。

像腾讯的风控，其依赖于庞大的微信、手 Q 生态体系的客户数据，日均调用量达 2000 亿次；京东的风控体系，涵盖零售、数科、物流、健康等线上线下多业务场景，跨多个领域且闭环；还有就是阿里的风控，相比京东，不仅有零售、数科、物流等，还有大文娱之类，场景更丰富。

那么为什么场景越丰富，相对来说风控的准确率越高呢？

这是因为风控的建设过程，其实就是一个不断完善用户画像的过程，而用户画像是建立风控的基础。一个用户画像的基础要素包括手机号、设备号、身份、IP、地址等，一些延展的信息还包括信贷记录、购物记录、履信记录、工作信息、社保信息等等。这些数据的收集，仅仅依靠单平台是无法做到的，这也是为什么风控的建立需要多平台、广业务、深覆盖，因为只有这样，才能够尽可能多地拿到用户数据。

有了这些数据，所谓的风控，其实就是针对某个用户，在不同的业务场景下，检查用户画像中的某些数据，是否触碰了红线，或者是某几项综合数据，是否触碰了红线。而有了完善的用户画像，那些黑产用户，在风控的照妖镜下，自然也就无处遁形了。

小结

这节课我们主要介绍了如何对抗黑产。黑产用户因为利益的引诱，通过外部工具、第三方软件甚至模拟请求的方式参与抢购活动，因为其速度更快、发出请求的频率更高，使得黑产用户获得了比普通用户更大的抢购成功率，这种行为不仅严重破坏了公平的抢购环境，同时也给秒杀系统带来了巨大的额外负担。

所以针对这种情形，我们也给出了几种应对的方案。像 Nginx 有条件限流机制，它可以直接有效地拦截针对接口的高频刷子请求；Token 机制确保了刷子流量无法跳过中间步骤直接下单；还有黑名单机制，配置合理的情况下，可以彻底拦截刷子流量。最后我们还简单地介绍了风控机制，即使黑产藏得很深，依然能将其揪出。

当然每个机制虽有其优点，但也都有其不足。像 Nginx 有条件限流，无法完全拦截刷子；黑名单机制不能抓到隐形的黑产用户；风控虽然可以抓到隐形黑产，但体系的搭建非常困

难。所以我们往往需要将各种机制做灵活的组合使用，从多维度为秒杀活动的进行保驾护航！

思考题

如果我既想做到严格针对用户 ID 的防刷，又不想使用 Redis，该如何实现呢？

以上就是这节课的全部内容，期待你的思考和答案，我们评论区见！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 0 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 08 | 化骨绵掌：降级、热点和容灾处理

精选留言 (1)

写留言

一个卖火柴的老男人

2021-10-15

嗯，很喜欢

展开